METHOD AND DEVICE FOR MANAGING LOCK OF OBJECT

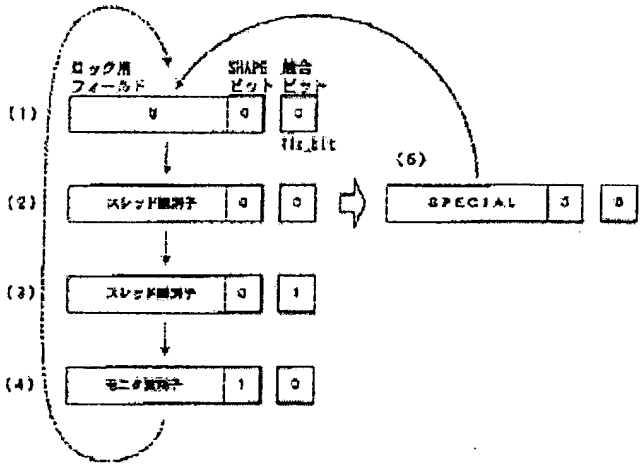
Patent number:	JP2001188685
Publication date:	2001-07-10
Inventor:	ONODERA TAMIYA
Applicant:	IBM
Classification:	
- international:	G06F9/46
- european:	G06F9/46R2
Application number:	JP19990371730 19991227
Priority number(s):	JP19990371730 19991227

Also published as:

US2001014905 (A1)

Abstract of **JP2001188685**

PROBLEM TO BE SOLVED: To provide a new composite locking method not to lower the processing speed of a high frequency path. SOLUTION: When no thread to lock an object exists 1, zero is stored in both of a field for lock and a competitive bit. After that, the object is locked (light weight lock) by a certain thread, an identifier of the thread is stored in the field for lock 2. If no lock is attempted by other threads before the lock is released by the thread with the thread identifier, SPECIAL is stored in the field for lock 5 and the processing is returned to 1. When the lock is attempted by other threads before the lock is released, competition in the light weight lock is generated, therefore, the competitive bit is established to record the competition 3. After that, the competitive bit is cleared when the lock is transferred to heavy weight lock 4 and the processing of 4 is transferred to 1, if possible.



(19)日本国特許庁（ＪＰ）　(12)公開特許公報（A）　(11)特許出願公開番号
特開2001－188685
（P2001－188685A）
(43)公開日　平成13年７月10日(2001.7.10)

(51)Int.Cl.'	識別記号	ＦＩ	ﾃｰﾏｺｰﾄﾞ(参考)
Ｇ０６Ｆ　9/46	３４０	Ｇ０６Ｆ　9/46	３４０Ｈ　5Ｂ０９８

審査請求　有　請求項の数18　ＯＬ　（全 22 頁）

(21)出願番号　特願平11－371730

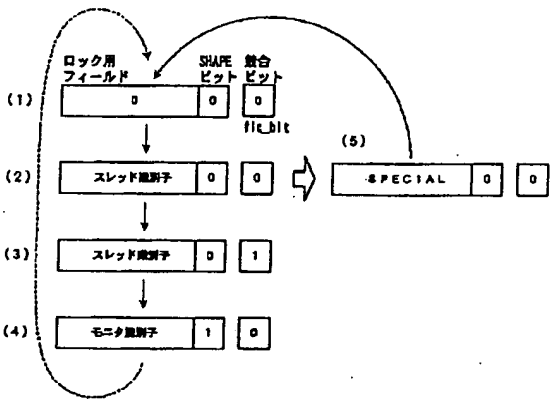
(22)出願日　平成11年12月27日(1999.12.27)

(71)出願人　390009531
インターナショナル・ビジネス・マシーンズ・コーポレーション
INTERNATIONAL BUSINESS MASCHINES CORPORATION
アメリカ合衆国10504、ニューヨーク州
アーモンク　（番地なし）
(74)復代理人　100079049
弁理士　中島　淳　（外５名）

最終頁に続く

(54)【発明の名称】　オブジェクトのロック管理方法及び装置

(57)【要約】
【課題】　高頻度パスの処理速度を低下させない、新規な複合ロック方法を提供する。
【解決手段】　オブジェクトをロックしているスレッドが存在しない場合（１）には、ロック用フィールド及び競合ビット共に０が格納される。その後、あるスレッドがそのオブジェクトをロック（軽量ロック）すると、そのスレッドの識別子がロック用フィールドに格納される（２）。もし、このスレッド識別子のスレッドがロックを解放するまでに他のスレッドがロックを試みなければ、ロック用フィールドにSPECIALを格納し（５）、（１）に戻る。ロックを解放するまでに他のスレッドがロックを試みると、軽量ロックにおける競合が発生したので、この競合を記録するため競合ビットを立てる（３）。その後、重量ロックに移行した際には、競合ビットはクリアされ（４）、可能であれば、（４）は（１）に移行する。



【特許請求の範囲】

【請求項１】　共有メモリモデルのシステムで、複数のスレッドが存在し得る状態において、オブジェクトに対応して設けられた記憶領域にロックの種類を示すビット及び第１の種類のロックに対応してロックを獲得したスレッドの識別子又は第２の種類のロックの識別子を記憶することによりオブジェクトへのロックを管理する方法であって、

第１のスレッドが保持しているあるオブジェクトへのロックを第２のスレッドが獲得しようとした場合、前記あるオブジェクトの前記ロックの種類を示すビットが第１の種類のロックであることを示しているか判断するステップと、

前記第１の種類のロックであることを示している場合には、競合ビットを立てるステップと、

前記第１のスレッドが保持しているあるオブジェクトへのロックを解除する際に、前記ロックの種類を示すビットが前記第１の種類のロックであることを示しているか判断するステップと、

前記複数のスレッドの識別子と異なる特殊識別子を前記記憶領域に記憶するステップと、

前記記憶領域の同期命令を発行するステップと、

前記あるオブジェクトのロックを保持しているスレッドが存在しないことを前記記憶領域に記憶するステップと、

前記ロックの種類を示すビットが前記第１の種類のロックであることを示している場合には、前記競合ビットが立っているか判断するステップと、

前記競合ビットが立っていないと判断された場合には、他の処理を実施せずにロック解除処理を終了するステップと、

を含むロック管理方法。

【請求項２】　前記競合ビットが立っていると判断された場合には、オブジェクトへのアクセスの排他制御と所定の条件が成立した場合のスレッドの待機操作及び待機しているスレッドへの通知操作とを可能にする機構の排他制御状態に前記第１のスレッドが移行するステップと、

待機しているスレッドへの通知操作を前記第１のスレッドが実行するステップと、

前記所定の条件が非成立でかつ前記特殊識別子が記憶されているとき、前記あるオブジェクトのロックを保持しているスレッドが存在せずかつ、ロックの種類を示すビットが第１の種類のロックになるまで前記第２のスレッドが繁忙待機するステップと、

前記第１のスレッドが前記排他制御状態から脱出するステップと、

をさらに含む請求項１に記載のロック管理方法。

【請求項３】　前記第１の種類のロックとは、オブジェクトに対してロックを実施するスレッドの識別子を当該オブジェクトに対応して記憶することによりロック状態を管理するロック方式である、請求項１に記載のロック管理方法。

【請求項４】　前記第２の種類のロックとは、オブジェクトへのアクセスを実施するスレッドをキューを用いて管理するロック方式である、請求項１記載のロック管理方法。

【請求項５】　共有メモリモデルのシステムで、複数のスレッドが存在し得る状態において、オブジェクトに対応して設けられた記憶領域にロックの種類を示すビット及び第１の種類のロックに対応してロックを獲得したスレッドの識別子又は第２の種類のロックの識別子を記憶することによりオブジェクトへのロックを管理する装置であって、

第１のスレッドが保持しているあるオブジェクトへのロックを第２のスレッドが獲得しようとした場合、前記あるオブジェクトの前記ロックの種類を示すビットが第１の種類のロックであることを示しているか判断する手段と、

前記第１の種類のロックであることを示している場合には、競合ビットを立てる手段と、

前記第１のスレッドが保持しているあるオブジェクトへのロックを解除する際に、前記ロックの種類を示すビットが前記第１の種類のロックであることを示しているか判断する手段と、

前記複数のスレッドの識別子と異なる特殊識別子を前記記憶領域に記憶する手段と、

前記記憶領域の同期命令を発行する手段と、

前記あるオブジェクトのロックを保持しているスレッドが存在しないことを前記記憶領域に記憶する手段と、

前記ロックの種類を示すビットが前記第１の種類のロックであることを示している場合には、前記競合ビットが立っているか判断する手段と、

前記競合ビットが立っていないと判断された場合には、他の処理を実施せずにロック解除処理を終了する手段と、

を含むロック管理装置。

【請求項６】　前記競合ビットが立っていると判断された場合には、オブジェクトへのアクセスの排他制御と所定の条件が成立した場合のスレッドの待機操作及び待機しているスレッドへの通知操作とを可能にする機構の排他制御状態に前記第１のスレッドが移行する手段と、

待機しているスレッドへの通知操作を前記第１のスレッドが実行する手段と、

前記所定の条件が非成立でかつ前記特殊識別子が記憶されているとき、前記あるオブジェクトのロックを保持しているスレッドが存在せずかつ、ロックの種類を示すビットが第１の種類のロックになるまで前記第２のスレッドが繁忙待機する手段と、

前記第１のスレッドが前記排他制御状態から脱出する手

段と、
をさらに含む請求項5に記載のロック管理装置。
【請求項7】 前記第1の種類のロックとは、オブジェクトに対してロックを実施するスレッドの識別子を当該オブジェクトに対応して記憶することによりロック状態を管理するロック方式である、請求項5に記載のロック管理装置。
【請求項8】 前記第2の種類のロックとは、オブジェクトへのアクセスを実施するスレッドをキューを用いて管理するロック方式である、請求項5記載のロック管理装置。
【請求項9】 共有メモリモデルのシステムで、複数のスレッドが存在し得る状態において、オブジェクトに対応して設けられた記憶領域にロックを示すビットを記憶し、オブジェクトへのアクセスを実施するスレッドのキューを記憶することによりオブジェクトへのロックを管理する方法であって、
第1のスレッドが保持しているあるオブジェクトへのロックを第2のスレッドが獲得しようとした場合、前記あるオブジェクトの前記ロックを示すビットがロックを示しているか判断するステップと、
前記ロックを示しているビットが立っている場合には、前記あるオブジェクトへのアクセスを実施するスレッドのキューの個数情報を変化させて記憶するステップと、
前記第2のスレッドをキューとして記憶することにより、前記あるオブジェクトへのアクセスの待機操作および通知による復帰操作する機構の制御状態に前記第2のスレッドが移行するステップと、
前記第1のスレッドが保持しているあるオブジェクトへのロックを解除する際に、前記ロックを示すビットを、前記あるオブジェクトのロックを示していることを前記記憶領域に記憶するステップと、
前記キューとして記憶されたスレッドが存在しているか判断するステップと、
前記キューとして記憶されたスレッドが存在していることを示している場合には、待機しているスレッドへの通知操作を実行する通知状態に前記第1のスレッドが移行するステップと、
前記第1のスレッドが前記通知状態から脱出するステップと、
を含むロック管理方法。
【請求項10】 前記ロックを示しているビットが立っている場合には、前記あるオブジェクトへのアクセスを実施するスレッドのキューの個数情報を増加させて記憶しかつ、前記あるオブジェクトの前記ロックを示すビットがロックを示しているか判断するステップと、
前記ロックを示しているビットが立っていない場合には、前記あるオブジェクトへのアクセスを実施するスレッドのキューの個数情報を減少させて記憶した後に他の処理を実施せずにロック処理を終了するステップと、
をさらに含む請求項9に記載のロック管理方法。
【請求項11】 共有メモリモデルのシステムで、複数のスレッドが存在し得る状態において、オブジェクトに対応して設けられた記憶領域にロックを示すビットを記憶し、オブジェクトへのアクセスを実施するスレッドのキューを記憶することによりオブジェクトへのロックを管理する方法であって、
第1のスレッドが保持しているあるオブジェクトへのロックを第2のスレッドが獲得しようとした場合、前記あるオブジェクトの前記ロックを示すビットがロックを示しているか判断するステップと、
前記ロックを示しているビットが立っている場合には、前記あるオブジェクトへのアクセスを実施するスレッドのキューの個数情報を変化させて記憶した後に、前記記憶領域の同期命令を発行するステップと、
前記第2のスレッドをキューとして記憶することにより、前記あるオブジェクトへのアクセスの待機操作および通知による復帰操作する機構の制御状態に前記第2のスレッドが移行するステップと、
前記第1のスレッドが保持しているあるオブジェクトへのロックを解除する際に、前記ロックを示すビットを、前記ロックを示していること及び示していないことと異なる識別子を前記記憶領域に記憶するステップと、
前記記憶領域の同期命令を発行するステップと、
前記あるオブジェクトのロックを示していないことを前記記憶領域に記憶するステップと、
前記キューとして記憶されたスレッドが存在しているか判断するステップと、
前記キューとして記憶されたスレッドが存在していることを示している場合には、待機しているスレッドへの通知操作を実行する通知状態に前記第1のスレッドが移行するステップと、
前記第1のスレッドが前記通知状態から脱出するステップと、
を含むロック管理方法。
【請求項12】 前記ロックを示しているビットが立っている場合には、前記あるオブジェクトへのアクセスを実施するスレッドのキューの個数情報を増加させて記憶しかつ、前記あるオブジェクトの前記ロックを示すビットがロックを示しているか判断するステップと、
前記ロックを示しているビットが立っていない場合には、前記あるオブジェクトへのアクセスを実施するスレッドのキューの個数情報を減少させて記憶した後に他の処理を実施せずにロック処理を終了するステップと、
をさらに含む請求項11に記載のロック管理方法。
【請求項13】 前記ロックを示しているビットが立っている場合でかつ、前記ロックを示していること及び示していないことと異なる識別子を前記記憶領域に記憶している場合、前記あるオブジェクトのロックを保持しているスレッドが存在せずかつ、ロックを示すビットがロ

ックを示していない状態になるまで前記第２のスレッドが繁忙待機するステップと、
をさらに含む請求項１２に記載のロック管理方法。
【請求項１４】　共有メモリモデルのシステムで、複数のスレッドが存在し得る状態において、オブジェクトに対応して設けられた記憶領域にロックを示すビットを記憶し、オブジェクトへのアクセスを実施するスレッドのキューを記憶することによりオブジェクトへのロックを管理する装置であって、
第１のスレッドが保持しているあるオブジェクトへのロックを第２のスレッドが獲得しようとした場合、前記あるオブジェクトの前記ロックを示すビットがロックを示しているか判断する手段と、
前記ロックを示しているビットが立っている場合には、前記あるオブジェクトへのアクセスを実施するスレッドのキューの個数情報を変化させて記憶する手段と、
前記第２のスレッドをキューとして記憶することにより、前記あるオブジェクトへのアクセスの待機操作および通知による復帰操作する機構の制御状態に前記第２のスレッドが移行する手段と、
前記第１のスレッドが保持しているあるオブジェクトへのロックを解除する際に、前記ロックを示すビットを、前記あるオブジェクトのロックを示していることを前記記憶領域に記憶する手段と、
前記キューとして記憶されたスレッドが存在しているか判断する手段と、
前記キューとして記憶されたスレッドが存在していることを示している場合には、待機しているスレッドへの通知操作を実行する通知状態に前記第１のスレッドが移行する手段と、
前記第１のスレッドが前記通知状態から脱出する手段と、
を含むロック管理装置。
【請求項１５】　前記ロックを示しているビットが立っている場合には、前記あるオブジェクトへのアクセスを実施するスレッドのキューの個数情報を増加させて記憶しかつ、前記あるオブジェクトの前記ロックを示すビットがロックを示しているか判断する手段と、
前記ロックを示しているビットが立っていない場合には、前記あるオブジェクトへのアクセスを実施するスレッドのキューの個数情報を減少させて記憶した後に他の処理を実施せずにロック処理を終了する手段と、
をさらに含む請求項１４に記載のロック管理装置。
【請求項１６】　共有メモリモデルのシステムで、複数のスレッドが存在し得る状態において、オブジェクトに対応して設けられた記憶領域にロックを示すビットを記憶し、オブジェクトへのアクセスを実施するスレッドのキューを記憶することによりオブジェクトへのロックを管理する装置であって、
第１のスレッドが保持しているあるオブジェクトへのロックを第２のスレッドが獲得しようとした場合、前記あるオブジェクトの前記ロックを示すビットがロックを示しているか判断する手段と、
前記ロックを示しているビットが立っている場合には、前記あるオブジェクトへのアクセスを実施するスレッドのキューの個数情報を変化させて記憶した後に、前記記憶領域の同期命令を発行する手段と、
前記第２のスレッドをキューとして記憶することにより、前記あるオブジェクトへのアクセスの待機操作および通知による復帰操作する機構の制御状態に前記第２のスレッドが移行する手段と、
前記第１のスレッドが保持しているあるオブジェクトへのロックを解除する際に、前記ロックを示すビットを、前記ロックを示していること及び示していないことと異なる識別子を前記記憶領域に記憶する手段と、
前記記憶領域の同期命令を発行する手段と、
前記あるオブジェクトのロックを示していないことを前記記憶領域に記憶する手段と、
前記キューとして記憶されたスレッドが存在しているか判断する手段と、
前記キューとして記憶されたスレッドが存在していることを示している場合には、待機しているスレッドへの通知操作を実行する通知状態に前記第１のスレッドが移行する手段と、
前記第１のスレッドが前記通知状態から脱出する手段と、
を含むロック管理装置。
【請求項１７】　前記ロックを示しているビットが立っている場合には、前記あるオブジェクトへのアクセスを実施するスレッドのキューの個数情報を増加させて記憶しかつ、前記あるオブジェクトの前記ロックを示すビットがロックを示しているか判断する手段と、
前記ロックを示しているビットが立っていない場合には、前記あるオブジェクトへのアクセスを実施するスレッドのキューの個数情報を減少させて記憶した後に他の処理を実施せずにロック処理を終了する手段と、
をさらに含む請求項１６に記載のロック管理装置。
【請求項１８】　前記ロックを示しているビットが立っている場合でかつ、前記ロックを示していること及び示していないことと異なる識別子を前記記憶領域に記憶している場合、前記あるオブジェクトのロックを保持しているスレッドが存在せずかつ、ロックを示すビットがロックを示していない状態になるまで前記第２のスレッドが繁忙待機するステップと、
をさらに含む請求項１７に記載のロック管理装置。
【発明の詳細な説明】
【０００１】
【発明の属する技術分野】本発明は、オブジェクトのロック管理方法及び装置にかかり、特に、複数のスレッドが存在し得る状態における、オブジェクトのロック管理

方法及び装置に関する。

【0002】

【従来の技術】複数のスレッドが動作するプログラムでオブジェクトへのアクセスを同期させるには、アクセスの前にオブジェクトをロック（lock）し、次にアクセスを行い、アクセスの後にアンロック（unlock）するようにプログラムのコードは構成される。このオブジェクトのロックの実装方法としては、スピンロック及びキューロック(サスペンドロックともいう)がよく知られている。また、最近ではそれらを組み合わせたもの（以下、複合ロックという）も提案されている。以下、それぞれについて簡単に説明する。

【0003】（1）スピンロック

```
10 /* ロック */
20 while (compare_and_swap(&;obj ->;lock,0,thread_id())==0)
30     yield();
40 /* objへのアクセス */
   . . .
50 /* アンロック */
60 obj ->;lock=0;
```

【0005】上記表1から理解されるように、第20行及び第30行でロックを行っている。ロックが獲得できるまで、yield()を行う。ここで、yield()とは、現在のスレッドの実行を止め、スケジューラに制御を移すことである。通常、スケジューラは、他の実行可能なスレッドから1つを選び走らせるが、いずれまた、スケジュラは、もとのスレッドを走らせることになり、ロックの獲得に成功するまで、while文の実行が繰り返される。yieldが存在していると、単にＣＰＵ資源の浪費だけでなく、実装がプラットフォームのスケジューリング方式に依存せざるを得ないため、期待どおりに動作するプログラムを書くことが困難になる。第20行におけるwhile文の条件であるcompare_and_swapは、オブジェクトobjに用意されたフィールドobj->;lockの内容と、0とを比較して、その比較結果が真であればスレッドのＩＤ（thread_id()）をそのフィールドに書き込むものである。よって、オブジェクトobjに用意されたフィールドに0が格納されている場合には、ロックしているスレッドが存在しないことを表している。よって、第60行でアンロックする場合にはフィールドobj->;lockに0を格納する。なお、このフィールドは例えば1ワードであるが、スレッド識別子を格納するのに十分なビット数であればよい。

【0006】（2）キューロック

キューロックとは、オブジェクトへのアクセスを実施するスレッドをキューを用いて管理するロック方式である。キューロックにおいては、スレッドTがオブジェクトobjのロックに失敗した場合、Tは自分自身をobjのキューに入れてサスペンドする。アンロックするコードには、キューが空か否かをチェックするコードが含まれ、空でなければキューから1つスレッドを取り出し、そのスレッドをリジュームする。このようなキューロックは、オペレーティング・システム（ＯＳ）のスケジューリング機構と一体になって実装され、ＯＳのＡＰＩ（Application Programming Interface）として提供されている。例えば、セマフォやＭｕｔｅｘ変数などが代表的なものである。キューロックにおいては、スペースオーバーヘッドはもはや1ワードでは済まず、十数バイトとなるのが普通である。また、ロックやアンロックの関数の内部では、キューという共有資源が操作されるため、何らかのロックが獲得され又は解放されている点にも注意する必要がある。

【0007】（3）複合ロック

マルチ・スレッド対応のプログラムは、マルチ・スレッドで実行されることを考慮して共有資源へのアクセスはロックにより保護するように書かれる。しかし、例えばマルチ・スレッド対応ライブラリがシングル・スレッドのプログラムから使用されるような場合もある。また、マルチ・スレッドで実行されてもロックの競合がほとんど発生しない場合もある。実際、Ｊａｖａ（Sun Microsystems社の商標）のプログラムの実行履歴によると、多くのアプリケーションにおいて、オブジェクトへのアクセスの競合はほとんど発生していないという報告もある。

【0008】よって、「ロックされていないオブジェクトにロックをかけ、アクセスし、アンロックする」は高頻度に実行されるパスであると考えられる。このパスは、スピンロックでは極めて効率よく実行されるが、キューロックでは時間的にも空間的にも効率が悪い。一方、高頻度ではないとはいえ、競合が実際に発生した場

スピンロックとは、オブジェクトに対してロックを実施するスレッドの識別子を当該オブジェクトに対応して記憶することによりロック状態を管理するロック方式である。スピンロックでは、スレッドTがオブジェクトobjのロック獲得に失敗した場合、すなわち他のスレッドＳが既にオブジェクトobjをロックしている場合、ロックに成功するまでロックを繰り返す。典型的には、compare#and#swap のようなアトミックなマシン命令（不可分命令）を用いて、次のようにロック又はアンロックする。

【0004】

【表1】

合、スピンロックではＣＰＵ資源が無益に消費されてしまうが、キューロックではそのようなことはない。

【０００９】複合ロックの基本的なアイデアは、スピンロックのような処理が簡単なロック（軽量ロックと呼ぶ）とキューロックのような処理が複雑なロック（重量ロックと呼ぶ）をうまく組み合わせて、上記の高頻度パスを高速に実行しつつ、競合時の効率も維持しようというものである。具体的に言えば、最初に軽量ロックでのロックを試み、軽量ロックで競合した場合重量ロックに遷移し、それ以降は重量ロックを使用するものである。

【００１０】この複合ロックでは、スピンロックと場合と同様に、オブジェクトにはロック用のフィールドがあり、「スレッド識別子」又は「重量ロック識別子」の値、及び、いずれの値を格納しているかを示すブール値が格納される。

【００１１】ロックの手順は以下のとおりである。

１）アトミックな命令（例えば、compare_and_swap）で軽量ロック獲得を試みる。成功すればオブジェクトへのアクセスを実行する。失敗した場合、すでに重量ロックになっているか、又は軽量ロックのままだが他のスレッドがロックをかけているのかのいずれかであることが分かる。

２）既に重量ロックになっていれば、重量ロックを獲得する。

３）軽量ロックで競合した場合、軽量ロックを獲得した上で重量ロックへ遷移し、これを獲得する（以下の説明では、inflate関数において実行される。）

【００１２】複合ロックには、３）における「軽量ロックの獲得」でyieldするか否かで２種類の実装がある。これらを詳しく以下に説明する。なお、ロック用のフィールドは１ワードとし、さらに簡単のため「スレッド識別子」又は「重量ロック識別子」は常に０以外の偶数であるとし、ロック用のフィールドの最下位ビットが０ならば「スレッド識別子」、１ならば「重量ロック識別子」が格納される。

【００１３】複合ロックの例１

軽量ロックの獲得において、yieldする複合ロックの場合である。ロック関数は上の手順に従って以下のように書くことができる。

【表２】

```
10: void lock(obj){
20:     if (compare_and_swap(&;obj ->;lock, 0, thread_id())
30:         return;
40:     while (! (obj ->;lock &; FAT_LOCK)){
50:
60:         if (compare_and_swap (&;obj ->;lock, 0, thread_id())){
70:             inflate(obj);
80:             return;
90:         }
92:         yield();
100:    }
110:    fat_lock(obj ->;lock)
120:    return;
130: }
140:
150: void unlock (obj){
160:     If (obj->;lock==thread_id())
170:        obj->;lock=0;
180:     else
190:         fat_unlock(obj->;lock);
200: }
220: void inflate(obj){
230:     obj->;lock= alloc_fat_lock() ¦ FAT_LOCK;
240:     fat_lock(obj->;lock);
250: }
```

【００１４】表２に示された擬似コードは、第１０行から第１３０行までがロック関数、第１５０行から第２００行までがアンロック関数、第２２０行から第２５０行までがロック関数で用いられるinflate関数を示している。ロック関数内では、第２０行で軽量ロックが試みられる。もしロックが獲得されれば、当該オブジェクトへのアクセスを実行する。そして、アンロックする場合には、第１６０行でオブジェクトのロック用フィールドにスレッド識別子が入力されているので、第１７０行においてそのフィールドに０を入力する。このように高頻度

パスはスピンロックと同じで高速に実行することができる。一方、第２０行でロックを獲得できない場合には、第４０行でｗｈｉｌｅ文の条件であるロック用フィールドの最下位ビットであるＦＡＴ＿ＬＯＣＫビットとロック用フィールドをビットごとにＡＮＤした結果が０であるか、すなわちＦＡＴ＿ＬＯＣＫビットが０であるか（より詳しく言うと軽量ロックであるか）判断される。もし、この条件が満たされていれば、第６０行にて軽量ロックを獲得するまでyieldする。軽量ロックを獲得した場合には、第２２０行以降のinflate関数を実行する。inflate関数では、ロック用フィールドobj->;lockに重量ロック識別子及び論理値１であるＦＡＴ＿ＬＯＣＫビット入力する（第２３０行）。そして、重量ロックを獲得する（第２４０行）。もし、第４０行で既にＦＡＴ＿ＬＯＣＫビットが１である場合には、直ぐに重量ロックを獲得する（第１１０行）。重量ロックのアンロックは第１９０行にて行われる。なお、重量ロックの獲得及び重量ロックのアンロックは、本発明とはあまり関係ないので説明を省略する。

【００１５】この表２ではロック用フィールドの書き換えは常に軽量ロックを保持するスレッドにより実施される点に注意されたい。これは、アンロックでも同じである。yieldが発生するのは、軽量ロックでの競合時に限定されている。

【００１６】複合ロックの例２

軽量ロックの獲得において、yieldしない複合ロックの例を示す。軽量ロックが競合した場合にはウエイト（wait）する。軽量ロック解放時には、ウエイトしているスレッドに通知（notify）しなければならない。このウエイト及び通知のためには、条件変数やモニタあるいはセマフォを必要とする。以下の例ではモニタを使用して説明する。

【００１７】

【表３】

```
10: void lock (obj){
20:     if (compare_and_swap (&;obj->;lock, 0, thread_id())
30:        return;
40:     monitor_enter (obj);
50:     while (!(obj->;lock, &; FAT_LOCK)){
60:         if (compare_and_swap(&;obj->;lock, 0, thread_id()){
70:            inflate(obj);
80:            monitor_exit(obj);
90:            return;
100:        } else
110:           monitor_wait(obj);
120:    }
130:    monitor_exit(obj);
140:    fat_lock(o->;lock);
150:    return;
160: }
180: void unlock (obj)
190:     if (obj->;lock == thread_id()) {
200:        obj->;lock=0;
210:        monitor_enter(obj);
220:        monitor_notify(obj);
230:        monitor_exit(obj);
240:    } else
250:        fat_unlock(obj->;lock);
260: }
280: void inflate (obj) {
290:     obj->;lock = alloc_fat_lock() ¦ FAT_LOCK
300:     fat_lock(obj->;lock);
310:     monitor_notify_all(obj);
320: }
```

【００１８】モニタとは、Ｈｏａｒｅによって考案された同期機構であり、オブジェクトへのアクセスの排他制御（enter及びexit）と所定の条件が成立した場合のスレッドの待機操作（wait）及び待機しているスレッドへの通知操作（notify 及びnotify_all）とを可能にする機構である（Hoare, C.A.R. Monitors: An operatingsy

stem structuring concept. CommunicationS of ACM 17, 10 (Oct. 1974), 549-557 参照)。高だか1つのスレッドがモニタにエンタ (enter) することが許される。スレッドTがモニタmにエンタしようとした時、あるスレッドSが既にエンタしているならば、Tは少なくともSがmからイグジット (exit) するまで待たされる。このように排他制御がなされる。また、モニタmにエンタ中のスレッドTは、ある条件の成立を待つため、モニタmでウエイト (wait) することができる。具体的には、Tは陰にmよりイグジットしサスペンドする。陰にmよりイグジットすることにより、別のスレッドがモニタmにエンタすることができる点に注意されたい。一方、モニタmにエンタ中のスレッドSは、ある条件を成立させた後に、モニタmに通知 (notify) することができる。具体的には、モニタmでウエイト中のスレッドのうちのひとつUを起こす (wake up) する。それにより、Uはリジュームし、モニタmに陰にエンタしようとする。ここで、Sがmにエンタ中であるから、Uは少なくともSがmからイグジットするまで待たされる点に注意されたい。また、モニタmでウエイト中のスレッドが存在しない場合には、何も起こらない。notify_allは、ウエイト中のスレッドを全て起こす点を除いて、notifyと同じである。

【0019】表3において、第10行乃至第160行はロック関数、第180行乃至第260行はアンロック関数、第280行乃至320行はinflate関数を示している。ロック関数で複合ロックの例1と異なる点は、第40行でモニタにエンタする点、軽量ロックで競合した場合にyieldせずにウエイトする点 (第110行)、重量ロックに遷移した際 (第80行) 及び重量ロックに遷移したことが確認された際 (第130行) にはモニタからイグジットする点である。ここで、第130行ではモニタからイグジットし、第140行で重量ロックを獲得している点に注意されたい。

【0020】アンロック関数で複合ロックの例1と異なる点は、第210行乃至第230行においてモニタにエンタし、モニタで通知をし、モニタをイグジットする処理を実施している点である。これは、yieldをやめてモニタにおけるウエイトにしたためである。inflate関数では、notify_allが追加されている。これもyieldをやめてモニタにおけるウエイトにしたためである。なお、第290行は、alloc_fat_lock()で得られる重量ロック識別子と論理値1にセットされたFAT_LOCKビットをOR操作して、ロック用フィールドに入力する操作を示している。

【0021】表3を見れば、yieldは消滅しているが、アンロック時にウエイトしているスレッドがいるかもしれないので、通知 (notify) という作業が入り、高頻度パスの性能が低下している。また、空間効率的には、モニタ又はモニタと同等な機能が余分に必要になっているが、重量ロックに遷移した後には不要になる。言いかえれば、モニタと重量ロックとは別に用意する必要がある。

【0022】

【発明が解決しようとする課題】ところで、論理的にメモリを共有する共有メモリモデルによるアーキテクチャでは、プロセッサとメモリが対称をなした対称型マルチプロセッサとよばれる共有メモリモデルによるシステム (以下、SMPシステムという) が知られている。このSMPシステムでは、命令レベル並列実行やメモリシステム最適化のため、メモリ操作(ReadおよびWrite)の順序は、用いるハードウェアによって変更される。すなわち、プログラムJを実行するプロセッサP1によるメモリ操作に関して、他のプロセッサP2の観測順序は、プログラムJの指定順序と同一であるとは限らない。例えば、IBM社のPowerPC、DEC社のAlpha、Sun社のSolaris RMO等の先進的なアーキテクチャでは、Read->;Read、Read->;Write、Write->;Read、Write->;Writeの全てにおいてプログラムの順序を保証していない。

【0023】しかしながら、プログラムによっては、プログラムの順序に従った観測が必要な場合もある。このため、上記アーキテクチャは、何れも、何らかのメモリ同期命令を提供している。例えば、PowerPCアーキテクチャは、メモリ同期命令としてSYNC命令を有している。プログラマはこの命令を陽 (直接的に) に用いることにより、ハードウェアによるメモリ操作の並べ替えを制限することができる。但し、メモリ同期命令は一般に高負荷であるので、多用することは好ましくない。

【0024】SMPシステムにおいてプログラムの順序に従った観測が必要となる処理の一例を、次に示す。

複合ロックの例3

【表4】

```
1: void lock(Object* obj){
2:  /* 軽量モードでのロック獲得 */
3:
4:  if (compare_and_swap(&;obj->;lock, 0, thread_id()))
5:   return; /* 成功 */
6:
7:  /* 失敗: モニタ突入し、重量モードに遷移 */
8:  MonitorId mon=obtain_monitor(obj);
9:  monitor_enter(mon);
```

```
10:  while ((obj->;lock &; FAT_LOCK)==0){
11:   set_flc_bit(obj);
12:
13:   if (compare_and_swap(&;obj->;lock, 0, thread_id()))
14:    inflate(obj, mon);
15:   else
16:    monitor_wait(mon);
17:  }
18: }
19:
20: void unlock(Object* obj){
21:  if ((obj->;lock &; FAT_LOCK)==0){ /* 軽量モード */
22:   MEMORY_BARRIER();
23:   obj->;lock=0;
24:   MEMORY_BARRIER();
25:   if (test_flc_bit(obj)){ /* 通常は失敗 */
26:    MonitorId mon=obtain_monitor(obj);
27:    monitor_enter(mon);
28:    if (test_flc_bit(obj))
29:     monitor_notify(mon);
30:    monitor_exit(mon);
31:   }
32:  }
33:  else { /* 重量モード */
34:   Word lockword=obj->;lock;
35:   if (no thread waiting on obj)
36:    if (better to deflate)
37:     obj->;lock=0; /* 軽量モードに遷移 */
38:   monitor_exit(lockword &;  FAT_LOCK);
39:  }
40: }
41:
42: void inflate (Object* obj, MonitorId mon) {
43:     clear_flc_bit(obj);
44:     monitor_notify_all (mon);
45:     obj->;lock= (Word) mon ¦ FAT_LOCK;
46: }
47:
48:
49: MonitorId obtain_monitor(Object* obj){
50:     Word word=obj->;lock;
51:     MonitorID mon;
52:     if (word &; FAT_LOCK)
53:          mon = word &; FAT_LOCK;
54:     else
55:          mon = lookup_monitor(obj);
56:     return mon;
57: }
```

【００２５】上記表のコードにおける特徴は、次の点である。なお、ここでは、高頻度パスの処理速度を低下させないために、競合ビット（ｆｌｃ＿ｂｉｔ）を新規に導入している。

【００２６】(1) オブジェクトヘッダ中の１つのフィールドをロック用に用いる。
(2) 軽量モードと重量モードの２つがあり、これらを区別するために形体ビット（ＦＡＴ＿ＬＯＣＫ）がある。なお、初期モードは軽量モードである。
(3) 軽量モードでは次のように動作する。軽量モードでは、ロックフィールドは、ロック状態ならばロックの保持者を、非ロック状態ならば０を格納する。スレッドＴは、ロックフィールドに自分の識別子を「原始的に」書き込むことによりロックを獲得する。スレッドＴは、ロックフィールドを「単純に(原始的ではなく)」ゼロクリアすることによりロックを解放する。
(4) 重量モードでは次のように動作する。このモードでは、ロックフィールドは、モニタ構造体への参照を格納している。重量モードでのロックの獲得解放は、モニタへの突入脱出に還元される。
(5) 軽量モードでのロック獲得時に競合が発生した場合、軽量モードから重量モードへ遷移（以下、ロックの膨張という）する。このとき、モニタ構造体が必要に応じ動的に割り当てられる。
(6) 重量モードでのロック解放時に、重量モードから軽量モードへ遷移（以下、ロックの収縮という）する場合がある。
【００２７】上記を図７を参照して説明する。図７に示したように、あるオブジェクトをロックしているスレッドが存在しない場合（（１）の場合）には、ロック用フィールド及び競合ビット共に０が格納される。その後、あるスレッドがそのオブジェクトをロック（軽量ロック）すると、そのスレッドの識別子がロック用フィールドに格納される（（２）の場合）。もし、このスレッド識別子のスレッドがロックを解放するまでに他のスレッドがロックを試みなければ（１）に戻る。ロックを解放するまでに他のスレッドがロックを試みると、軽量ロックにおける競合が発生したので、この競合を記録するため競合ビットを立てる（（３）の場合）。その後、重量ロックに移行した際には、競合ビットはクリアされる（（４）の場合）。可能であれば、（４）は（１）に移行する。なお、ロック用フィールドの最下位に軽量ロックと重量ロックのモードを表すビット（ＦＡＴ＿ＬＯＣＫビット）設けるようにしたが、最上位に設けるようにしても良い。
【００２８】次に、軽量モードでの動作および膨張処理について説明する。まず、lock関数の第４行目の原始命令により、軽量モードでのロックの獲得を試みる。軽量モードでかつ無競合ならば成功する。そうでなければ重量ロックを獲得、すなわち、モニタに突入し、膨張処理に入る。このとき、すでに重量モードならばwhile文の本体は実行されない。ここで、obtain_monitor関数は、オブジェクトに対応するモニタを返す関数である。対応関係はハッシュ表等で管理される。
【００２９】一方、unlock関数では、２１行目で形体ビットがテストされ、軽量モード時は第２２行～第２５行目を実行する。第２３行目はロック解放であるが、原始命令は使用していない。第２５行目のビットテストは、後述するが、ロックの膨張処理と関係し、無競合時は失敗し、if文の本体は実行されない。
【００３０】ところで、ＳＭＰシステム特有の処理が、第２２行目と第２４行目のＳＹＮＣ命令である。第２２行目のＳＹＮＣ命令は、ロック保持中に行なったメモリ操作命令をロック解放前に完了することを保証するもので、本複合ロックに限らず必要な処理である。一方、第２４行目のＳＹＮＣ命令は、第２３行目のロック解放と第２５行目のビットテストがプログラム順に完了することを強制するもので、本複合ロック独特のものである。
【００３１】本複合ロックの膨張処理の大きな特徴は、膨張処理で繁忙待機せずにモニタ待機することである。しかも、これを軽量モードでのロック解放に、原始命令を使用することなく実現しており、少なくともユニプロセッサでは理想的なロック法となっている。
【００３２】ここで、繁忙待機を止めてモニタ待機する際に、最大の難関は、通知保証、すなわち、「モニタ待機に入ったスレッドは必ず通知される」ことを保証することである。本複合ロックでは、ＦＬＣ(flat lock contention)ビットと呼ばれる、ロックフィールドとは別のワードに確保された１ビットを用いて、巧妙なプロトコルを構成することで、通知保証を実現している。これについて説明する。
【００３３】スレッドＴが、第１６行目でモニタ待機に入ったとする。これは、第１３行目の原始命令に失敗したことを意味する。この時刻をｔとする。ここで、プログラム順の完了が保証されるように第１０行～第１３行目が書かれているとすると、時刻ｔより前にＦＬＣビットはセットされている。
【００３４】一方、原始命令の失敗は、時刻ｔにおいて別のスレッドＳがロックを保持していること意味する。次の理由によりそれは軽量ロックである。本複合ロックは、常にモニタの保護下でモード遷移するようにコードが書かれている。スレッドＴは、第９行目の突入または第１６行目の待機からの復帰により、モニタに突入している。スレッドＴは、しかも、第１０行目で軽量モードであることを確認している。従って、第１２行目でもモードは変わらず軽量モードであることがわかる。
【００３５】時刻ｔでスレッドＳは軽量ロックを保持しているが、特に第２４行目のＳＹＮＣ命令を実行していない。従って、時刻ｔより後にＦＬＣビットはテストされる。
【００３６】以上のことにより、時刻ｔより前にスレッドＴはＦＬＣビットをセットし、時刻ｔより後にスレッドＳはＦＬＣビットをテストする。従って、スレッドＳは、第２５行目のテストに常に成功し、if文の本体を実

行し、スレッドTに対しモニタ通知する。すなわち、通知保証が達成されている。

【００３７】第２４行目のＳＹＮＣ命令がなければ、第２５行目のビットの読み出しは、第２３行目の書き込みより先に実行される可能性があり、ＦＬＣビットのテストが、原始命令の失敗時刻tより後だとは保証できない。したがって、第２４行目のＳＹＮＣ命令は、本複合ロックの正当性に不可欠なものである。

【００３８】このように、本複合ロックでは、ＳＭＰシステムで実装した場合、軽量モード無競合時のロック解放において、メモリ同期命令を２つ用いる必要がある。

【００３９】なお、重量ロックの解除では、第３３行に処理は移行する。第３４行では、lockwordという変数にロック用フィールドの内容を格納する。そして、モニタにおける待機状態（wait）にあるスレッドが他に存在しないかを判断する（第３５行）。もし、存在しない場合には、所定の条件を満たしているか判断する（第３６行）。所定の条件には、重量ロックから脱出しない方が良いような条件があればそのような条件を設定する。但し、本ステップは実行しなくてもよい。もし、所定の条件を満たしている場合には、ロック用フィールドobj->;lockを０にする（第３７行）。すなわち、ロックを保持しているスレッドが存在しないことをロック用フィールドに格納する。そして、変数lockwordのＦＡＴ＿ＬＯＣＫビット以外の部分に格納されたモニタ識別子のモニタからイグジットする（第３８行）。lockword &; FAT_LOCKは、ＦＡＴ＿ＬＯＣＫビットを反転させたものとlockwordとのビットごとのＡＮＤである。これにより、モニタにエンタしようとして待機していたスレッドが、モニタにエンタできるようになる。

【００４０】次に、モニタ識別子を獲得するobtain_monitor関数を説明する。この関数では、上記と同様に、lockwordという変数にロック用フィールドの内容を格納する（第５０行）。そして、モニタの識別子を格納する変数ｍｏｎを用意し（第５１行）、ＦＡＴ＿ＬＯＣＫビットが立っているか判断する（第５２行、word &; FAT_LOCK）。もし、ＦＡＴ＿ＬＯＣＫビットが立っているようであれば、変数ｍｏｎにlockwordのＦＡＴ＿ＬＯＣＫビット以外の部分を格納する（第５３行、lockword &; FAT_LOCK）。一方、ＦＡＴ＿ＬＯＣＫビットが立っていない場合には、関数lookup_monitor(obj)を実行する（第５５行）。この関数は、オブジェクトとモニタの関係を記録したハッシュ・テーブルを有していることを前提とし、基本的にはこのテーブルをオブジェクトobjについて検索して、モニタの識別子を獲得する。もし、必要があれば、モニタを生成し、そのモニタの識別子をハッシュ・テーブルに格納した後にモニタ識別子を返す。いずれにしても、変数ｍｏｎに格納されたモニタの識別子を返す。

【００４１】本発明の目的は、高頻度パスの処理速度を低下させない、新規な複合ロック方法を提供することである。

【００４２】

【課題を解決するための手段】上記目的を達成するために本発明は、軽量モード無競合時のロック解放において、メモリ同期命令を必要最小限、すなわち特殊識別子による先行解放と本解放の２段解放によってメモリ同期命令を減少させている。具体的には、共有メモリモデルのシステムで、複数のスレッドが存在し得る状態において、オブジェクトに対応して設けられた記憶領域にロックの種類を示すビット及び第１の種類のロックに対応してロックを獲得したスレッドの識別子又は第２の種類のロックの識別子を記憶することによりオブジェクトへのロックを管理する場合に、第１のスレッドが保持しているあるオブジェクトへのロックを第２のスレッドが獲得しようとした場合、前記あるオブジェクトの前記ロックの種類を示すビットが第１の種類のロックであることを示しているか判断するステップと、前記第１の種類のロックであることを示している場合には、競合ビットを立てるステップと、前記第１のスレッドが保持しているあるオブジェクトへのロックを解除する際に、前記ロックの種類を示すビットが前記第１の種類のロックであることを示しているか判断するステップと、前記複数のスレッドの識別子と異なる特殊識別子を前記記憶領域に記憶するステップと、前記記憶領域の同期命令を発行するステップと、前記あるオブジェクトのロックを保持しているスレッドが存在しないことを前記記憶領域に記憶するステップと、前記ロックの種類を示すビットが前記第１の種類のロックであることを示している場合には、前記競合ビットが立っているか判断するステップと、前記競合ビットが立っていないと判断された場合には、他の処理を実施せずにロック解除処理を終了するステップと、を実行する。

【００４３】このようにして、軽量モード無競合時のロック解放では、高価なメモリ同期命令を少なくとも２つ発行することなく、本発明のように２段解放によればメモリ同期命令を１つのみに減少させることができる。

【００４４】また、前記競合ビットが立っていると判断された場合には、オブジェクトへのアクセスの排他制御と所定の条件が成立した場合のスレッドの待機操作及び待機しているスレッドへの通知操作とを可能にする機構の排他制御状態に前記第１のスレッドが移行するステップと、待機しているスレッドへの通知操作を前記第１のスレッドが実行するステップと、前記所定の条件が非成立でかつ前記特殊識別子が記憶されているとき、前記あるオブジェクトのロックを保持しているスレッドが存在せずかつ、ロックの種類を示すビットが第１の種類のロックになるまで前記第２のスレッドが繁忙待機するステップと、前記第１のスレッドが前記排他制御状態から脱出するステップと、をさらに実行する。

【００４５】このように、待機しているスレッドへの通知操作を実行するステップと、前記所定の条件が非成立でかつ前記特殊識別子が記憶されているとき、前記あるオブジェクトのロックを保持しているスレッドが存在せずかつ、ロックの種類を示すビットが第１の種類のロックになるまでロック処理を待機する繁忙待機を採用する。

【００４６】なお、第１の種類のロックとは、オブジェクトに対してロックを実施するスレッドの識別子を当該オブジェクトに対応して記憶することによりロック状態を管理するロック方式である。また、第２の種類のロックとは、オブジェクトへのアクセスを実施するスレッドをキューを用いて管理するロック方式である。

【００４７】また、共有メモリモデルのシステムで、複数のスレッドが存在し得る状態において、オブジェクトに対応して設けられた記憶領域にロックを示すビットを記憶し、オブジェクトへのアクセスを実施するスレッドのキューを記憶することによりオブジェクトへのロックを管理する場合、第１のスレッドが保持しているあるオブジェクトへのロックを第２のスレッドが獲得しようとした場合、前記あるオブジェクトの前記ロックを示すビットがロックを示しているか判断するステップと、前記ロックを示しているビットが立っている場合には、前記あるオブジェクトへのアクセスを実施するスレッドのキューの個数情報を変化させて記憶するステップと、前記第２のスレッドをキューとして記憶することにより、前記あるオブジェクトへのアクセスの待機操作および通知による復帰操作する機構の制御状態に前記第２のスレッドが移行するステップと、前記第１のスレッドが保持しているあるオブジェクトへのロックを解除する際に、前記ロックを示すビットを、前記あるオブジェクトのロックを示していることを前記記憶領域に記憶するステップと、前記キューとして記憶されたスレッドが存在しているか判断するステップと、前記キューとして記憶されたスレッドが存在していることを示している場合には、待機しているスレッドへの通知操作を実行する通知状態に前記第１のスレッドが移行するステップと、前記第１のスレッドが前記通知状態から脱出するステップと、を実行する。

【００４８】このようにすれば、一般的なスピンサスペンドロックに対して、ロックとアンロックとの各々にアトミックなマシン命令（不可分命令）を必要としない。すなわち、ロックするときにのみ、アトミックなマシン命令を用いるのみで、アンロック時にはアトミックなマシン命令を用いることのない代入等の命令でよい。

【００４９】前記ロックを示しているビットが立っている場合には、前記あるオブジェクトへのアクセスを実施するスレッドのキューの個数情報を増加させて記憶しかつ、前記あるオブジェクトの前記ロックを示すビットがロックを示しているか判断するステップと、前記ロックを示しているビットが立っていない場合には、前記あるオブジェクトへのアクセスを実施するスレッドのキューの個数情報を減少させて記憶した後に他の処理を実施せずにロック処理を終了するステップと、をさらに実行することができる。

【００５０】また、共有メモリモデルのシステムで、複数のスレッドが存在し得る状態において、オブジェクトに対応して設けられた記憶領域にロックを示すビットを記憶し、オブジェクトへのアクセスを実施するスレッドのキューを記憶することによりオブジェクトへのロックを管理する場合に、第１のスレッドが保持しているあるオブジェクトへのロックを第２のスレッドが獲得しようとした場合、前記あるオブジェクトの前記ロックを示すビットがロックを示しているか判断するステップと、前記ロックを示しているビットが立っている場合には、前記あるオブジェクトへのアクセスを実施するスレッドのキューの個数情報を変化させて記憶した後に、前記記憶領域の同期命令を発行するステップと、前記第２のスレッドをキューとして記憶することにより、前記あるオブジェクトへのアクセスの待機操作および通知による復帰操作する機構の制御状態に前記第２のスレッドが移行するステップと、前記第１のスレッドが保持しているあるオブジェクトへのロックを解除する際に、前記ロックを示すビットを、前記ロックを示していること及び示していないことと異なる識別子を前記記憶領域に記憶するステップと、前記記憶領域の同期命令を発行するステップと、前記あるオブジェクトのロックを示していないことを前記記憶領域に記憶するステップと、前記キューとして記憶されたスレッドが存在しているか判断するステップと、前記キューとして記憶されたスレッドが存在していることを示している場合には、待機しているスレッドへの通知操作を実行する通知状態に前記第１のスレッドが移行するステップと、前記第１のスレッドが前記通知状態から脱出するステップと、を実行する。

【００５１】このようにすれば、一般的なスピンサスペンドロックに対して、ロックとアンロックとの各々にアトミックなマシン命令（不可分命令）を必要としない。さらに、同期命令を少なくとも、２つ用いる必要もない。すなわち、メモリをロックするときの前後で同期命令が必要であったが、本発明によれば、２段階の解除により１つの同期命令のみでよいことになる。

【００５２】この場合、前記ロックを示しているビットが立っている場合には、前記あるオブジェクトへのアクセスを実施するスレッドのキューの個数情報を増加させて記憶しかつ、前記あるオブジェクトの前記ロックを示すビットがロックを示しているか判断するステップと、前記ロックを示しているビットが立っていない場合には、前記あるオブジェクトへのアクセスを実施するスレッドのキューの個数情報を減少させて記憶した後に他の処理を実施せずにロック処理を終了するステップと、を

さらに実行することができる。

【００５３】また、前記ロックを示しているビットが立っている場合でかつ、前記ロックを示していること及び示していないことと異なる識別子を前記記憶領域に記憶している場合、前記あるオブジェクトのロックを保持しているスレッドが存在せずかつ、ロックを示すビットがロックを示していない状態になるまで前記第２のスレッドが繁忙待機するステップと、をさらに実行することもできる。

【００５４】以上述べた本発明の処理は、専用の装置として実施することも、また、コンピュータのプログラムとして実施することも可能である。さらに、このコンピュータのプログラムは、ＣＤ－ＲＯＭやフロッピー・ディスク、ＭＯ（Magneto-optic）ディスクなどの記憶媒体、又はハードディスクなどの記憶装置に記憶される。

【００５５】

【発明の実施の形態】以下、図面を参照して本発明の実施の形態の一例を詳細に説明する。本実施の形態はＳＭＰシステムに本発明を適用したものである。

〔第１実施の形態〕図１には本発明の処理が実施されるコンピュータの例を示す。コンピュータ１０００は、ハードウエア１００と、ＯＳ（Operating System）２００、アプリケーション・プログラム３００を含む。ハードウエア１００は、ＣＰＵ（１又は複数）１１０及びＲＡＭ１２０等のメインメモリ、及びハードウェア資源にアクセスするための入出力インタフェース（Ｉ／Ｏインタフェース）１３０を含んでいる。ＯＳ２００は、カーネル側領域２００Ａとユーザ側領域２００Ｂから構成されており、ＡＰＩ（Application Programming Interface）２１０を含んでいる。また、ＯＳ２００は、ハードウエア１００とアプリケーション・プログラム３００との間の操作を可能にする、すなわち、アプリケーション・プログラム３００として動作する複数のスレッドを可能にする機能を有するスレッド・ライブラリ２２０を備えている。このスレッド・ライブラリ２２０はキューロックに必要な機能も提供する。また、アプリケーション・プログラム３００は、モニタ機能、本発明のロック及びアンロック機能を含む。また、データベース言語の場合には、データベース・マネイジメント・システム３３０をＯＳ２００上に設け、さらにその上でアプリケーション３４０を実行する場合もある。さらに、Ｊａｖａ言語の場合には、Ｊａｖａ　ＶＭ（Virtual Machine）３１０をＯＳ２００上に設け、さらにその上でアプレット又はアプリケーション３２０を実行する場合もある。アプレット又はアプリケーション３２０もマルチ・スレッドで実行され得る。Ｊａｖａ言語では、Ｊａｖａ　ＶＭ３１０に、モニタ機能、ロック及びアンロック機能が組み込まれる場合もある。また、Ｊａｖａ　ＶＭ（３１０）はＯＳ２００の一部として組み込まれる場合もある。また、コンピュータ１０００は補助記憶装置を有しない、所謂ネットワークコンピュータ等でもよい。

【００５６】（二段解放によるメモリ同期命令の削減）発明が解決しようとする課題の欄で説明したように、複合ロックの例３では、ＳＭＰシステムで実装した場合、軽量モード無競合時のロック解放において、高価なメモリ同期命令を２つ発行しなくてはならない。そこで、本実施の形態では、特殊識別子による先行解放と本解放の２段解放によってメモリ同期命令を１つのみに減少させている。

【００５７】まず、どのスレッドにも割り当てられることのない識別子を1つ選び、unlock関数の軽量モード時の手順を、特殊識別子による先行解放、メモリ同期命令、本解放とする。本実施の形態では、先行解放のための特殊識別子としてSPECIALを導入する。

【００５８】図２に示したように、あるオブジェクトをロックしているスレッドが存在しない場合（（１）の場合）には、ロック用フィールド及び競合ビット共に０が格納される。その後、あるスレッドがそのオブジェクトをロック（軽量ロック）すると、そのスレッドの識別子がロック用フィールドに格納される（（２）の場合）。もし、このスレッド識別子のスレッドがロックを解放するまでに他のスレッドがロックを試みなければ、ロック用フィールドにSPECIALを格納し（（５）の場合）、（１）に戻る。ロックを解放するまでに他のスレッドがロックを試みると、軽量ロックにおける競合が発生したので、この競合を記録するため競合ビットを立てる（（３）の場合）。その後、重量ロックに移行した際には、競合ビットはクリアされる（（４）の場合）。可能であれば、（４）は（１）に移行する。なお、ロック用フィールドの最下位に軽量ロックと重量ロックのモードを表すビット（ＦＡＴ＿ＬＯＣＫビット）設けるようにしたが、最上位に設けるようにしても良い。

【００５９】この特殊識別子としてSPECIALを導入した処理を以下に示す。

【表５】

```
10: void unlock(Object* obj){
20:   if ((obj->;lock &; SHAPE_BIT)==0){ /* 軽量モード */
30:     obj->;lock=SPECIAL;
40:     MEMORY_BARRIER();
50:     obj->;lock=0;
60:     if (test_flc_bit(obj)){ /* 通常は失敗 */
70:       MonitorId mon=obtain_monitor(obj);
```

```
 80:       monitor_enter(mon);
 90:       if (test_flc_bit(obj))
100:         monitor_notify(mon);
110        monitor_exit(mon);
120:     }
130:   }
140:   else { /* 重量モード */
150:     Word lockword=obj->;lock;
160:     if (no thread waiting on obj)
170:       if (better to deflate)
180:         obj->;lock=0; /* 軽量モードに遷移 */
190:     monitor_exit(lockword &; FAT_LOCK);
200:   }
210: }
220:
230: void lock(Object* obj){
240:   /* 軽量モードでのロック獲得 */
250:   int unlocked=0;
260:   if (compare_and_swap_370(&;obj->;lock,&;unlocked,thread_id()))
270:     return; /* 成功 */
280:
290:  /* 失敗: モニタ突入し、重量モードに遷移 */
300:  MonitorId mon=obtain_monitor(obj);
310:  monitor_enter(mon);
320:  while ((obj->;lock &; SHAPE_BIT)==0){
330:    set_flc_bit(obj);
340:    unlocked=0;
350:    if (compare_and_swap_370(&;obj->;lock,&;unlocked,thread_id()))
360:      inflate(obj, mon);
370:    else if (unlocked==SPECIAL)
380:      ; /* 繁忙待機 */
390:    else
400:      monitor_wait(mon);
410:    }
420:  }
430:
440: void inflate (Object* obj, MonitorId mon) {
450:     clear_flc_bit(obj);
460:     monitor_notify_all (mon);
470:     obj->;lock= (Word) mon ¦ SHAPE;
480: }
490:
500:
510: MonitorId obtain_monitor(Object* o){
520:     Word lockword=obj->;lock;
530:     MonitorID mon;
540:     if (lockword &; SHAPE)
550:          mon = lockword &; SHAPE;
560:     else
570:          mon = lookup_monitor(o);
```

```
580:      return mon;
590: }
```

【0060】上記では、IBM SyStem/370で定義された　　る。
本来のcompare_and_swap_370を用いている。この関数co　　【表6】
mpare_and_swap()は、次の作業を原始的に行うものであ

```
100:  int compare_and_swap_370(Word* word, Word *old, Word new){
110:    if(*word==*old){
120:      *word=new; return 1; /* succeed */
130:    }else{
140:      *old=*word; return 0; /* fail */
150:    }
160:  }
```

【0061】なお、競合ビットは表4ではｆｌｃ＿ｂｉｔとして示されている。上記表5は、4つの部分からなる。ロック関数の部分（第220行乃至第420行）、アンロック関数の部分（第10行乃至第210行）、軽量ロックから重量ロックへの遷移であるinflate関数の部分（第440行乃至第480行）、及びモニタの識別子を獲得するobtain_monitor関数の部分（第510行乃至第590行）である。以下、表5の処理を詳細に説明する。なお、表5では、表4のＦＡＴ＿ＬＯＣＫビットに代えてＳＨＡＰＥビットとして示されている。

【0062】（1）ロック関数

第230行から始まったオブジェクトobjに対するロック関数の処理では、まず軽量ロックの獲得を試みる（第250行及び第260行）。この軽量ロックの獲得には、本実施の形態ではcompare_and_swapのようなアトミックな命令を用いる。この命令では、第1の引き数と第2の引き数が同じ値の場合、第3の引き数を格納するものである。ここでは、オブジェクトobjのロック用フィールドであるobj->;lockが0に等しい場合には、thread_id()によりスレッド識別子を獲得して、ロック用フィールドobj->;lockに格納する。図2の（1）から（2）への遷移を実施したものである。そして、必要な処理を実施するため、リターンする（第270行）。もし、オブジェクトobjのロック用フィールドであるobj->;lockが0に等しくない場合には、軽量ロックの獲得は失敗し、第300行に移行する。

【0063】次に、モニタ識別子を獲得するobtain_monitor(obj)関数の値をmonという変数に代入し（第300行）、スレッドはそのモニタの排他制御状態に移行しようとする。すなわちモニタ（monitor）にエンタ（enter）しようとする（第310行）。もし、排他制御状態に移行することができれば、以下の処理を実施し、もしできなかった場合には、できるまでこの段階で待つ。次に、while文の条件を判断する。すなわち、ロック用フィールドobj->;lockとＳＨＡＰＥビットのビットごとのＡＮＤを実施し、ＳＨＡＰＥビットが立っているか判断する（第320行）。ここでは、現在重量ロックに移行しているのか、軽量ロック中なのかを判断している。もし、ＳＨＡＰＥビットが立っていなければ（軽量ロック中）、この計算の結果は0となるから、while文以下の処理を実施する。一方、ＳＨＡＰＥビットが立っている場合（重量ロック中）、while文以下の処理を実施せずに、モニタにエンタした状態のままになる。このようにＳＨＡＰＥビットが立っている場合に、モニタにエンタできた場合には、重量ロックを獲得できたということを意味しており、このモニタからイグジット（exit）することなく（すなわち排他制御状態を脱出することなく）、このスレッドはオブジェクトに対する処理を実施する。

【0064】一方、第320行でＳＨＡＰＥビットが立っていないと判断された場合には、軽量ロックの競合が発生していることを意味するので、ｆｌｃ＿ｂｉｔをセットする（第330行、set_flc_bit(obj)）。これは、図2の（2）から（3）への遷移に相当する。そして、もう一度軽量ロックを獲得できるか判断する（第340行及び愛350行）。軽量ロックを獲得できる場合には軽量ロックから重量ロックへの遷移のためのinflate関数の処理を実施する（第360行）。一方、軽量ロックが獲得できずかつunlock変数がSPECIALである場合には、繁忙待機する。すなわち、本実施の形態では、繁忙待機を再導入している。これは、先行解放は観測されたが本解放は観測されていない場合であり、本解放が目前に迫っているときである。ＳＭＰシステムの場合、この時機では、繁忙待機したほうが好ましいためである。軽量ロックが獲得できずかつunlock変数がSPECIALでもない場合には、モニタの待機状態（wait）に移行する（第400行）。モニタの待機状態は、モニタから脱出してサスペンドするものである。このように、軽量ロックで競合が生じると、競合ビットであるｆｌｃ＿ｂｉｔがセットされ、軽量ロックを獲得できない場合には、モニタの待機状態に移行する。この待機状態に入ると、後にinflate関数の処理又はアンロックする際に通知（notify又はnotify_all）を受けることになる。

【0065】（2）inflate関数

次に、inflate関数の処理を説明する。ここではまず、競合ビットがクリアされる（第450行、clear_flc_bi

t)。そして、モニタの通知操作(monitor_notify_all)を実施する(第４６０行)。ここでは、待機状態の全てのスレッドに起きる(wake up)よう通知する。そして、ロック用フィールドObj->;lockに、モニタの識別子を格納した変数ｍｏｎとセットされたＳＨＡＰＥビットをビットごとにＯＲした結果を格納する(第４４０行、mon | SHAPE)。すなわち、図2の(３)から(４)の状態に遷移させたものである。これで軽量ロックから重量ロックへの遷移は完了する。なお、第３６０行の処理が終了すると、再度while文の条件をチェックすることになるが、既にＳＨＡＰＥビットが立っているので、この場合にはwhile文から脱出して、モニタにエンタしたままとなる。すなわち、while文の中の処理を実行しない。

【００６６】通知を受けた全てのスレッドは第４００行において陰にモニタにエンタしようとするが、モニタにエンタする前に待機することになる。これは、通知を行ったスレッドはアンロック処理を実施するまでモニタからイグジットしていないからである。

【００６７】(３)アンロック関数

次に、アンロック関数の処理について説明する。アンロック関数は軽量ロックのアンロックと、重量ロックのアンロックを取扱う。

【００６８】軽量ロックのアンロック

軽量ロックのアンロックでは、まず、ロック用フィールドobj->;lockとＳＨＡＰＥビットのビットごとのＡＮＤを計算し、その値が０であるか判断する(第２００行)。これは、ロック関数のwhile文の条件と同じであって(第３２０行)、軽量ロック中であるかどうか判断するものである。軽量ロック中である場合には、特殊識別子による先行解放を実施する(第３０行:obj->;lock=SPECIAL)。これは、図2の(２)から(５)への遷移に相当する。そして、メモリ同期命令(第４０行:MEMORY_BARRIER( ))を行った後、本解放のため、ロック・フィールドobj->;lockに０を格納する(第５０行)。これは、図2の(５)から(１)への遷移に相当する。

【００６９】このようにして、軽量モード無競合時のロック解放において、高価なメモリ同期命令を２つ発行することなく、２段解放によってメモリ同期命令を１つのみに減少させている。すなわち、本発明におけるメモリ同期命令がこの第４０行である。このようにして、unlock関数の軽量モード時の手順を、特殊識別子による先行解放、メモリ同期命令、本解放としている。これにより、ロックを保持しているスレッドが存在しないことが記録される。そして、競合ビットが立っているか判断する(第６０行、test_flc_bit)。軽量ロックで競合が生じていなくとも、第６０行のみは実施しなければならない。競合ビットが立っていない場合には、アンロック処理を終了する。

【００７０】一方、競合ビットが立っている場合には、ロック関数の第３００行及び第３１０行と同様に、変数ｍｏｎにモニタの識別子を格納し(第７０行)、当該モニタ識別子のモニタにエンタしようとする(第８０行)。すなわち、そのスレッドはモニタの排他制御状態に入ろうとする。もしモニタにエンタできた場合には、もう一度、競合ビットが立っていることを確認し(第９０行)、もし立っていれば、モニタにおいて待機状態のスレッドの１つに起動を通知する(第１００行、monitor_notify(mon))。なお、モニタにエンタできない場合には、モニタにエンタできるまで待機する。そして通知を行ったスレッドは、モニタの排他制御状態から脱出する(第１１０行、monitor_exit(mon))。

【００７１】第１００行で通知を受けたスレッドは、第４００行で陰にモニタにエンタする。そして第９０行に戻りその処理を実施する。通常、第１００行で通知を受けたスレッドは、通知を行ったスレッドがモニタの排他制御状態を脱出した後にモニタの排他制御状態に入り、競合ビットを立てた後に、軽量ロックを獲得し、inflate関数の処理を実施することにより重量ロックに遷移する。

【００７２】本実施の形態では、メモリ同期命令は１つであるが、上記の複合ロックの例３と略同様の議論を展開することにより、通知保証が達成されていることがわかる。すなわち、スレッドＴがモニタ待機に入ったとする。スレッドＴが原始命令に失敗した時刻をｔとすると、時刻ｔより以前にＦＬＣビットはセットされている。ここで、時刻ｔにおけるロックフィールドの値は、SPECIALでもない点に注意されたい。

【００７３】そして、別のスレッドＳが時刻ｔで軽量ロックを保持している。しかも、時刻ｔにおけるロックフィールドの値がSPECIALでもないことから、時刻ｔでスレッドＳはＳＹＮＣ命令を実行していない。すなわち、ＦＬＣビットのテストは時刻ｔより後である。特に、本解放とＦＬＣビットのテストが逆順に実行されたとしてもそうである。以上のことにより通知保証は達成されている。

【００７４】なお、本実施の形態では、繁忙待機が再導入されているが、それは極めて限定的なものである。のみならず、ＳＭＰシステムにおいては、このような繁忙待機はむしろ有効ですらある。その理由は、本実施の形態において繁忙待機するのは、先行解放は観測されたが本解放は観測されていない場合である。すなわち、本解放が目前に迫っているときである。ＳＭＰシステムの場合、こうした時機では、モニタ待機しコンテキスト切替えを行うより、繁忙待機したほうが得策である。

【００７５】〔第２実施の形態〕本発明は、一般的なスピンサスペンド・ロック(スピンロックとキューロックとを組み合わせた複合ロック)に対して有効である。すなわち、次に示すように、一般的なスピンサスペンド・ロックで表すことができる。以下の説明では、この一般

的なスピンサスペンド・ロックを、一般複合ロックという。なお、スピンサスペンド・ロックは広く応用されており、ＯＳ／２のcritical section、ＡＩＸのpthreadsライブラリのmutex変数の実装においても使用されている。しかし、無競合時の性能を考えた場合、既存のアルゴリズムは、ロック獲得及び解放にそれぞれ原始命令を必要とするのに対して、本実施の形態の一般複合ロックは、ロック獲得においてのみ原始命令を必要とするものである。なお、本実施の形態は、上記の実施の形態と略同様の構成のため、同一部分には同一符号を付して詳細な説明を省略する。

【００７６】まず、本実施の形態の処理のうちロック処理を説明する。図３に示すように、ステップ２０００において原始命令を用いた軽量ロックの獲得を試みて、次のステップ２０１０において獲得に成功したか否かを判断し、獲得に成功したときは、本ルーチンを終了する。獲得に失敗したときは、すでに他のスレッドによりロックされているため、サスペンド・モードへ移行し、次のステップ２０２０において、pthreadsライブラリが提供するものと同様のセマンティクスを有するmutex変数を用いたフィールドをロックする。次のステップ２０３０では、待機スレッドの数を表すフィールドの値を増加する。すなわち、現在のスレッドを、待機しているスレッドに追加すべく表明する。次のステップ２０４０では、再度軽量ロックの獲得を試みる。獲得に成功したときは、ステップ２０６０においてフィールドの値を減少した後にmutex変数を用いたフィールドをアンロックする。一方、獲得に失敗したときは、ステップ２０８０において獲得を試みたスレッドをキューとして待機しステップ２０４０へ戻る。

【００７７】次に、アンロック処理を説明する。図４に示すように、ステップ２１００においてロック用のフィールドを解放する状態にした後に次のステップ２１１０において待機スレッドの数を表すフィールドの値を獲得する。待機スレッドがないときはフィールドの値は「０」であるため、この場合には本ルーチンを終了する。一方、待機スレッドが存在するときは、ステップ２１２０で肯定され、ステップ２１３０においてmutex変数を用いたフィールドをロックする。次のステップ２１４０では再度待機スレッドの数を表すフィールドの値を獲得して次のステップ２１５０において再度待機スレッドが存在するか否かを判断する。このときに、待機スレッドがないときは、ステップ２１７０においてmutex変数を用いたフィールドをアンロックした後に本ルーチンを終了する。一方、待機スレッドが存在するときは、ステップ２１６０において待機しているスレッドを読み出して（通知し）、ステップ２１７０においてmutex変数を用いたフィールドをアンロックした後に本ルーチンを終了する。

【００７８】上記の処理の流れに沿った一般複合ロックのアルゴリズムを以下に示す。

【表７】

```
 10:  typedef struct {
 20:    int    lock;   /* initially, UNLOCKED */
 30:    int    wcount;   /* initially, 0 */
 40:    mutex_t mutex;
 50:      condvar_t condvar
 60:  }tsk_t;
 70:
 80:  void tsk_lock(tsk_t *tsk){
 90:    if(compare_and_swap(&;tsk->;lock; UNLOCKED, LOCKED))
100:      return; /* ok */
110:    tsk_suspend(tsk);
120:  }
130:
140:  void tsk_unlock(tsk_t *tsk){
150:    tsk->;lock= UNLOCKED;
160:      if(tsk->;wcount)
170:        tsk_resume(tsk);
180:  }
190:
200:  void tsk_suspend(tsk_t *tsk){
210:    mutex_lock(&;tsk->;mutex);
220:    tsk->;wcount++;
230:    while(1){
240:      if(compare_and_swap(&;tsk->;lock; UNLOCKED, LOCKED)){
```

```
250:          tsk->;wcount--;
260:          break;
270:      }
280:      else
290:          condvar_wait(&;tsk->;condvar, &;tsk->;mutex);
300:    }
310:    mutex_unlock(&;tsk->;mutex);
320: }
330:
340: void tsk_resume(tasuki_t *tsk){
350:   mutex_lock(&;tsk->;mutex);
360:   if(tsk->;wcount)
370:     condvar_signal(&;tsk->;condvar);
380:   mutex_unlock(&;tsk->;mutex);
390: }
```

【００７９】本アルゴリズムでは、pthreadsライブラリが提供するものと同じセマンティクスを有するmutex変数とcondvar変数を用いて、tsk_suspend関数とtsk_resume関数を記述しているが、基本的にアルゴリズムの説明のためである。一般複合ロックは、スレッドライブラリ上に作成されるものではなく、むしろカーネル空間の中でよりカスタマイズされた形で作成されるべきものである。

【００８０】なお、condvar_wait()関数は、条件付変数であり、第１引数を変数として待機しかつmutex変数を解除するものである。これに対応して、condvar_signal()関数は、通知を行うものである。

【００８１】表６において、第１０行乃至第５０行は、用いるデータの構造、第８０行乃至第１２０行はロック関数、第１４０行乃至第１８０行はアンロック関数、第２００行乃至３２０行はサスペンド関数、第３４０行乃至３９０行はレジューム関数を示している。

【００８２】表６から理解されるように、ロック関数は、単純なスピンロックを含むこととなる。また、アンロック関数は、現在、フィールドtsk->;wcountのテストを含むことを示す。これはロックを獲得しているスレッドがサスペンド・ロックに後退していた他のスレッド用のロック解除にいくらかの動作をとることを必要とするか否かを示している。しかし、第１６０行のｉｆ文の条件が成立しない限り、tsk_resume関数は実行されない。これは、単純なテストを行うものであって、重要な処理を行うものではない。従って、表６のアルゴリズムは、最も速くスピンロックしているアルゴリズムと比較すると単純なテストを１つ加えているだけである。

【００８３】また、tsk_suspend関数は、呼んでいるスレッドがスピン・ロックを得ようとするwhile loopを含んでいる。これはspin-waitループでなくsuspend-waitループである。すなわち、第２９０行で待機している何れのスレッドも休眠状態が解除されることを約束しなければならない。このために、フィールドtsk->;wcount内の現在待機しているスレッドの数の、情報を獲得し続けることになる。カウンタ（tsk->;wcount）は、mutexの保護の下で増加そして減少する。従って、同じ保護の下にカウンタを検査することによって、どれだけのスレッドが待機しているか否かを、正しく確認することができる。

【００８４】ところで、アンロック関数は、いかなる保護もないカウンタをチェックして、カウンタの間違った値を読み込む場合があるが本実施の形態では、先と同様の推論を展開することにより、休眠状態の解除通知は保証される。

【００８５】（ＳＭＰシステムへの具体的適用）ところで、一般複合ロックをＳＭＰシステムすなわち先進的ＳＭＰマシンで実装する場合、複合ロックの例３と同様の問題に遭遇する。すなわち、無競合時のロック解放に、メモリ同期命令を２つ発行しなければならない。具体的には、表６の第１５０行：「tsk->;lock = UNLOCKING;」の前後である。この問題は上記実施の形態と同様の処理によって、解決できる。このＳＭＰシステムへ適用した一般複合ロックを、本実施の形態では、ＳＭＰ複合ロックと呼ぶ。

【００８６】まず、ロック処理を説明する。図５に示すように、原始命令を用いた軽量ロックの獲得を試み（図３のステップ２０００）、獲得に成功したか否かを判断し（図３のステップ２０１０）、獲得に成功したときは、本ルーチンを終了する。獲得に失敗したときは、すでに他のスレッドによりロックされているため、サスペンド・モードへ移行し、pthreadsライブラリが提供するものと同様のセマンティクスを有するmutex変数を用いたフィールドをロックする（図３のステップ２０２０）。次に、待機スレッドの数を表すフィールドの値を増加する（図３のステップ２０３０）。すなわち、現在のスレッドを、待機しているスレッドに追加すべく表明する。次に、ステップ２２００において、メモリ同期命令（MEMORY_BARRIER()）を発行した後に、次のステップ

２２１０において変数unlockedを解放する。このメモリ同期命令は、上述したＳＹＮＣ命令と同様に、ロック保持中に行ったメモリ操作命令をロック解放前に完了することを保証する機能を有している。

【００８７】この後に、再度軽量ロックの獲得を試みる（図3のステップ２０４０）。獲得に成功したときは、フィールドの値を減少（図3のステップ２０６０）した後にmutex変数を用いたフィールドをアンロックする（図3のステップ２０７０）。一方、獲得に失敗したときは、ステップ２２３０において変数unlockedが充填されているか否かを判断し、肯定判断の場合にはそのままステップ２０４０へ戻り、否定判断の場合には獲得を試みたスレッドをキューとして待機（図3のステップ２０８０）しステップ２０４０へ戻る。

【００８８】次に、アンロック処理を説明する。図6に示すように、ステップ２４００においてロック用のフィールドに先行解放を示す値を代入した後にステップ２４１０においてメモリ同期命令を発行する。次のステップ２４２０ではロックフィールドを解放し、次のステップ２４３０において待機スレッドの数を表すフィールドの値を獲得する。待機スレッドがないときはフィールドの値は「０」であるため、この場合には本ルーチンを終了する。一方、待機スレッドが存在するときは、図4のステップ２１３０以降の処理と同様に、mutex変数を用いたフィールドをロックし、再度待機スレッドの数を表すフィールドの値を獲得して再度待機スレッドが存在するか否かを判断する。このときに、待機スレッドがないときは、mutex変数を用いたフィールドをアンロックし、待機スレッドが存在するときは、待機しているスレッドを読み出し（通知し）た後に、mutex変数を用いたフィールドをアンロックした後に本ルーチンを終了する。

【００８９】上記の処理の流れに沿ったＳＭＰ複合ロックのアルゴリズムを以下に示す。ここでは、上述のIBM SyStem／370で定義された本来のcompare_and_swap_370を用いる。この関数compare_and_swap_370を用いるとした場合、tsk_unlock関数、tsk_suspend関数は次のようになる。他の２つの関数は同じである。

【表８】

```
500:  void tsk_unlock_smp(tsk_t *tsk){
510:    tsk->;lock = UNLOCKING;
520:    MEMORY_BARRIER();
530:    tsk->;lock = UNLOCKED
540:
550:    if(tsk->;wcount)
560:      tsk_resume(tsk);
570:  }
580:
590:  void tsk_suspend_smp(tsk_t *tsk){
600:    mutex_lock(&;tsk->;mutex);
610:    tsk->;wcount++;
620:    MEMORY_BARRIER();
630:    while(1){
640:      int unlocked=UNLOCKED;
650:      if(compare_and_swap_370(&;tsk->;lock; UNLOCKED, LOCKED)){
660:          tsk->;wcount—;
670:          break;
680:      }elseif(unlocked == UNLOCKING){
690:          ;/* spin-wait */
700:      }else
710:          condvar_wait(&;tsk->;condvar, &;tsk->;mutex);
720:    }
730:    mutex_unlock(&;tsk->;mutex);
740:  }
```

【００９０】このように、本実施の形態では、割り当てられてない識別子を1つ選び、unlock関数の軽量モード時の手順を、特殊識別子による先行解放、メモリ同期命令、本解放とする。本実施の形態では、先行解放のための特殊識別子としてUNLOCKINGを導入している。すなわち、アンロック関数において、フィールドtsk->;lockの値をＬＯＣＫＥＤあるいはＵＮＬＯＣＫＥＤ以外のＵＮＬＯＣＫＩＮＧに一旦設定し、メモリバリヤした後に、アンロックしている。従って、特殊識別子による先行解放と本解放の２段解放で１つのみのメモリ命令での処理を可能にしている。

【００９１】

【発明の効果】以上説明したように本発明によれば、軽量モード無競合時のロック解放において、すなわち特殊識別子による先行解放と本解放の２段解放によってメモリ同期命令を必要最小限に減少させることができる、という効果がある。

【図面の簡単な説明】

【図１】本発明の処理が実施されるコンピュータの一例を示す図である。

【図２】本発明のモードの遷移、並びに各モードにおけるロック用フィールド（ＳＨＡＰＥビットを含む）及び競合ビットの状態を説明するための図であり、（１）はロックなし、（２）は軽量ロックで競合なし、（３）は軽量ロックで競合あり、（４）は重量ロック、（５）は特殊識別子による先行解放の状態を示す。

【図３】本発明の一般複合ロックのロック処理の流れを示すフローチャートである。

【図４】本発明の一般複合ロックのアンロック処理の流れを示すフローチャートである。

【図５】本発明のＳＭＰ複合ロックのロック処理の流れを示すフローチャートである。

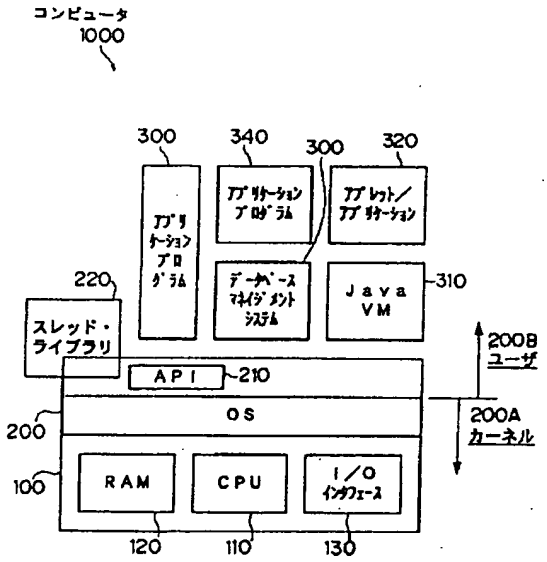
【図６】本発明のＳＭＰ複合ロックのアンロック処理の流れを示すフローチャートである。

【図７】複合ロックの例３のモードの遷移、並びに各モードにおけるロック用フィールド（ＦＡＴ＿ＬＯＣＫビットを含む）及び競合ビットの状態を説明するための図であり、（１）はロックなし、（２）は軽量ロックで競合なし、（３）は軽量ロックで競合あり、（４）は重量ロックの状態を示す。

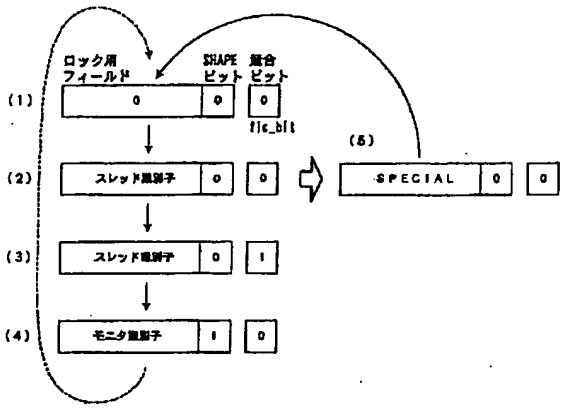
【符号の説明】

１０００	コンピュータ
１００	ハードウエア
２００	ＯＳ
２００Ａ	カーネル領域
２００Ｂ	ユーザ領域
２１０	ＡＰＩ
２２０	スレッドライブラリ
３００	アプリケーション・プログラム
３１０	Ｊａｖａ　ＶＭ
３２０	Ｊａｖａ　アプレット／アプリケーション
３３０	データベースマネイジメントシステム

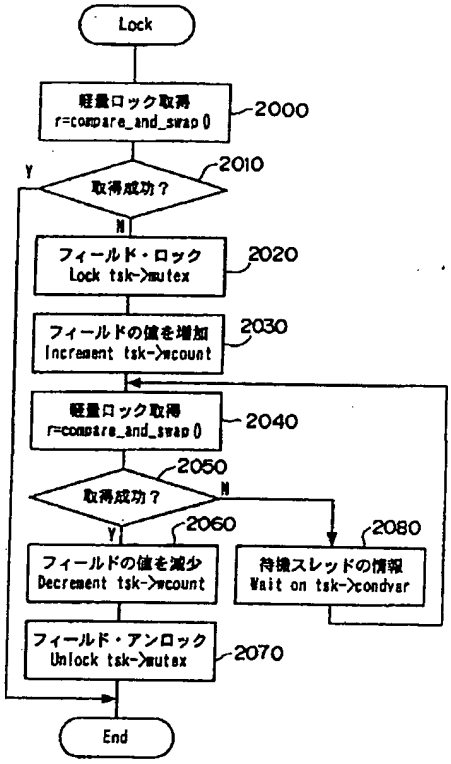
【図１】



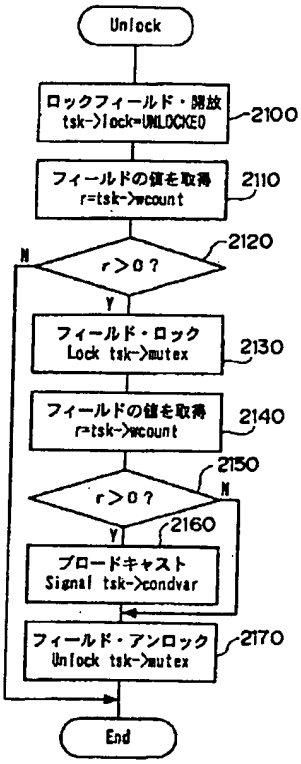
【図２】



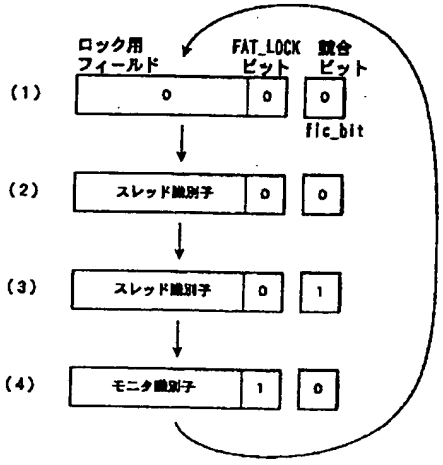
【図3】


Lock
軽量ロック取得
r=compare_and_swap()
2000
取得成功？
2010
Y
N
フィールド・ロック
Lock tsk->mutex
2020
フィールドの値を増加
Increment tsk->wcount
2030
軽量ロック取得
r=compare_and_swap()
2040
取得成功？
2050
N
Y
フィールドの値を減少
Decrement tsk->wcount
2060
待機スレッドの情報
Wait on tsk->condvar
2080
フィールド・アンロック
Unlock tsk->mutex
2070
End

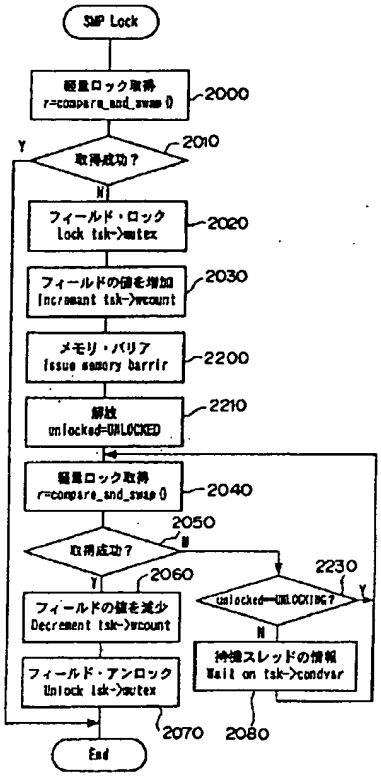
【図4】


Unlock
ロックフィールド・開放
tsk->lock=UNLOCKED
2100
フィールドの値を取得
r=tsk->wcount
2110
r>0？
2120
N
Y
フィールド・ロック
Lock tsk->mutex
2130
フィールドの値を取得
r=tsk->wcount
2140
r>0？
2150
N
Y
ブロードキャスト
Signal tsk->condvar
2160
フィールド・アンロック
Unlock tsk->mutex
2170
End

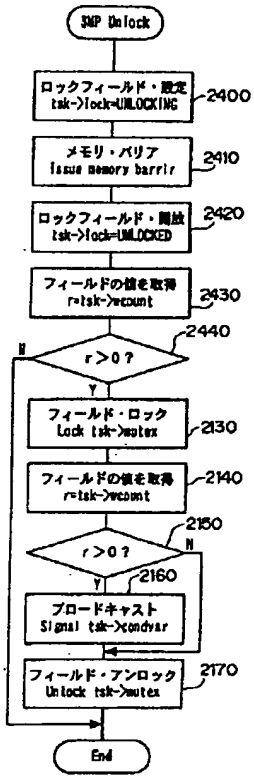
【図7】


ロック用
フィールド
FAT_LOCK
ビット
競合
ビット
(1)
0
0
0
flc_bit
(2)
スレッド識別子
0
0
(3)
スレッド識別子
0
1
(4)
モニタ識別子
1
0

【図5】



【図6】



フロントページの続き

(72)発明者　小野寺　民也
神奈川県大和市下鶴間1623番地14　日本アイ・ビー・エム株式会社　東京基礎研究所内

Fターム(参考)　5B098 AA03 GA05 GB13 GD03 GD16